# STL相关

list底层是链表，unordered_map底层是哈希表，map底层是红黑树

什么是数据结构：可以看成是计算机存储数据的方式，然后把这些相互之间存在某种特定关系的数据元素归为一类，就是一种数据结构。(一般有两种存储结构，顺序存储、链式存储)

什么是算法：是指解决一个问题的方法或者说是步骤，一个算法的优劣可以用空间复杂度与时间复杂度来衡量。

# 0.各数据结构对比

## 0.1数组

数组是指将相同数据类型的元素按一定顺序排列的集合， 在使用前需要提前申请内存空间，如果不知道所需的空间大小，预先申请的空间可能过大造成浪费，所以数组的空间利用率较低。查询元素时，可根据数组索引可直接查询，因为数组中元素是连续存放的，添加和删除元素时，需要对其他元素进行移动，所以**插入和删除元素的效率低。**

优点：查询快,通过索引直接查找，时间复杂度O(1)

缺点：在中间部位增删需要移动后面的数据，时间复杂度为O(n)；数组大小固定；只能存储一种类型的数据

使用场景：查询多，增加和删除少的情况

## 0.2链表

链表是由一系列离散的节点组成的，每个节点包含一个数据域和一个指针域，通过各节点的指针将离散的节点连接起来，内存利用率比较高。因为链表的空间离散，所以可以任意插入和删除元素，但是当要访问某个位置的数据时，需要从第一个位置开始往后遍历，所以**查询效率低**。

| 操作 | 数组 | 链表 |
| --- | --- | --- |
| 随机访问 | O(1) | O(N) |
| 头部插入元素 | O(N) | O(1) |
| 头部删除元素 | O(N) | O(1) |
| 尾部插入元素 | O(1) | O(1) |
| 尾部删除元素 | O(1) | O(1) |



优点：**插入快，删除快**，只需要修改元素指针指向就可以

缺点：**查找慢**，因为是离散存储各个结点的，所以需要从第一个元素一个个结点去遍历

使用场景：少查询，需要频繁插入或删除的情况

## 0.3 栈

栈是一种只允许在栈顶(表尾)进行插入和删除操作的数据结构(**线性表**)。

线性表：由一些相同类型的数据元素组成的有限序列。(表中每个元素具有顺序性，线性表中存在唯一一个被称为第一个的元素和被称为最后一个的元素)

优点：栈是先进后出的存取方式，所以添加元素很快

缺点：只有栈顶元素可以被外界使用不能遍历，存取栈顶元素之外的其他元素会比较慢

使用场景：使用场景：①实现递归 ②字符串逆序 ③符号是否匹配等

栈**可由数组或者单向链表实现**，是一种ADT（抽象数据类型），入栈和出栈的时间复杂度都是O(1)

# 0.4 队列

队列是一种只允许在一端(队尾)进行插入操作，另一端(队头)只允许删除操作的数据结构。

优点：提供先进先出的存取方式，添加新元素速度快

缺点：队列中只有队头和队尾才可以被外界使用，因此队列不允许有遍历行为。

队列也**可以由数组和双端链表实现**，也是一种ADT，出队和入队的时间复杂度是O(1)
优先级队列的插入操作需要 O(n)的时间，而删除操作则需要O(1) 的时间

使用场景：多线程阻塞队列管理非常有用

# 0.5 堆

堆从内存视角来看，它是一个连续存放的数组，而从逻辑视角来看它是一颗顺序存储的完全二叉树，堆又分为大**根堆和小根堆**：大根堆是指在完全二叉树当中，所有子树的根节点值都要大于等于它的左右子树节点的值。小根堆是指完全二叉树当中，所有子树的根节点值都要小于等于它的左右子树节点的值。

# 0.6 哈希表

哈希表是根据键值(key)直接访问在内存存储位置的一种数据结构。

原理：是通过函数映射的方式把关键字和数据存储的位置关联起来，这样就能通过关键字直接找到数据，不需要通过数据比较的方式进行查找，所以哈希表的**查找效率很高**。

常见哈希函数：

①直接定址法：即 f(key) = a*key + b，f表示哈希函数，a、b是常量，key是键值。

②除数留余法：即 f(key) = key % p，p 表示容器数量，这种方式通常用在将数据存放到指定容器中。

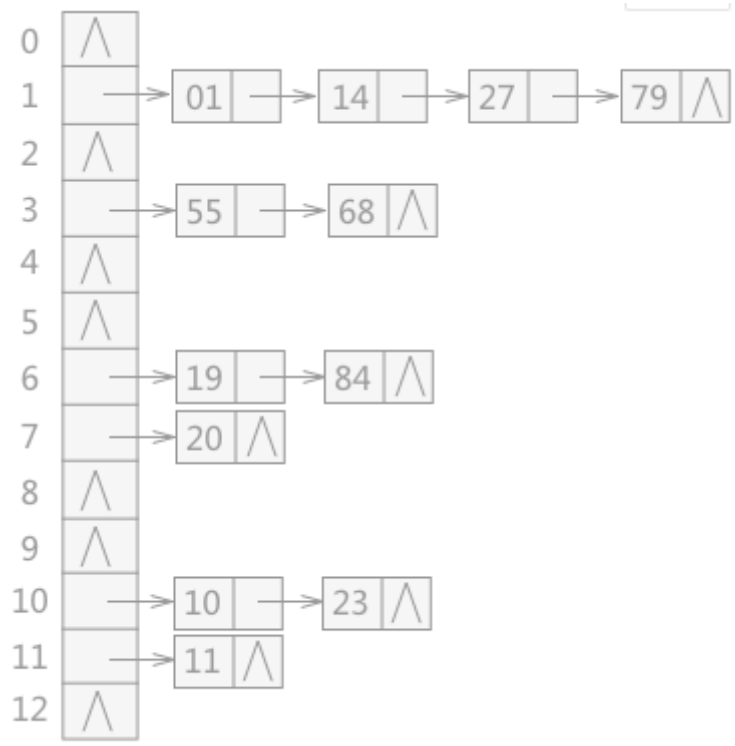
优点：如果关键字已知则存取、插入删除都很快，都是O(1)。
缺点：不稳定，如果出现哈希冲突，存取、插入删除的时间复杂度是O(n)
应用场景：哈希表适用于那种查找性能要求高，数据元素之间无逻辑关系要求的情况，在海量数据处理中有广泛应用。

使用场景：负载均衡
对于同一个客户端上的请求，尤其是已登录用户的请求，需要将其会话请求都路由到同一台机器，以保证数据的一致性，这可以借助哈希算法来实现，通过用户 ID 尾号对总机器数（服务器数）取模（取多少位可以根据机器数定），将结果值作为机器编号。

**哈希冲突：**当关键字集合很大时，关键字值不同的元素可能会映射到哈希表的同一地址上，这种现象称为哈希冲突。

处理哈希冲突的办法：

1.链接地址法（开链法）：将所有产生冲突的关键字所对应的数据存储在同一个单链表中，并将单链表的头指针存在关键字所对应的value中，所以查找、插入和删除主要在这个链表中进行。链地址法适用于经常进行插入和删除的情况。




2.再哈希法：同时构造多个不同的哈希函数，当第一个哈希函数计算的哈希地址发生冲突时，再使用第二个哈希函数计算哈希地址，直到不冲突为止。

3.开放定址法（再散列法）：当哈希冲突发生时，使用某种探测算法在散列表中寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到。

4.建立一个公共溢出区：建立两张表，一张为基本表，另一张溢出表。基本表存储没有发生冲突的数据，当关键字由哈希函数生成的哈希地址产生冲突时，就将数据填入溢出表。

**哈希表满了怎么办：**当哈希表数据存储量超过一定程度

(1)再次新建一个新的哈希表。新表的尺寸一般是旧表的2倍。

(2)将之前的数据再次通过哈希计算搬进表里

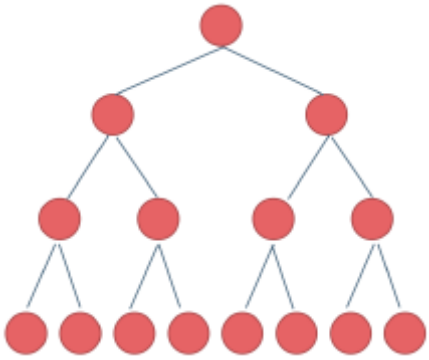
(3)丢弃掉原来的表

# 0.7树

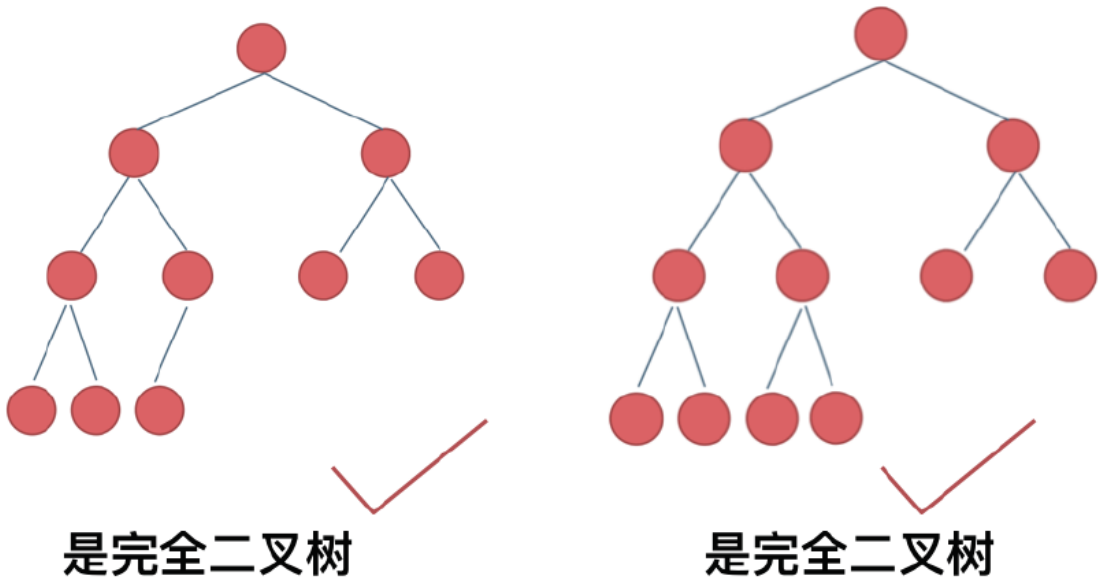
## 1. 二叉树

二叉树：树的每个节点最多只能有两个子节点

**满二叉树**：一颗二叉树只有度为0的节点和度为2的节点，并且度为0的节点都在同一层上，它就是满二叉树。(节点的度就是这个节点的孩子数量)，也可以说是深度为k，有(2^k)-1个节点



**完全二叉树**：除了最底层节点可能没填满，其余每层节点数都达到最大值。并且最下面一层的节点都集中在该层左边。




**二叉搜索树**：若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值； 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值； 它的左、右子树也分别为二叉搜索树。

树的效率：查找节点的时间取决于这个节点所在的层数，每一层最多有2n-1个节点，总共N层共有2n-1个节点，那么时间复杂度为O(log2N)。（N表示的是二叉树节点的总数，而不是层数）。增删查的时间复杂度都是O(log2N)

优点：如果是平衡二叉树，那么查找，插入，删除都快

**2. 平衡二叉搜索树(AVL)**

(插入、删除、查找 O(logn))

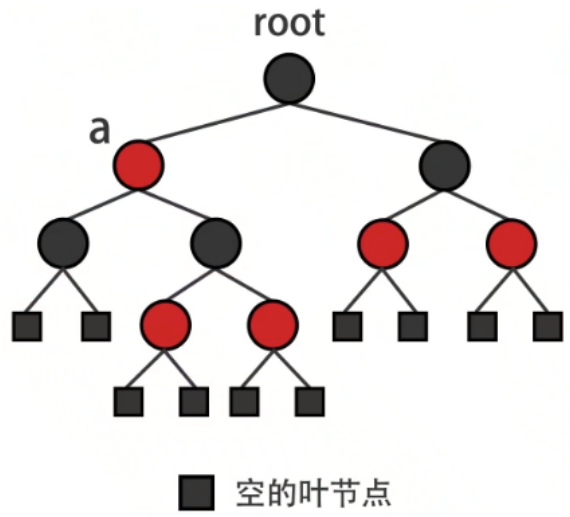
AVL树:规定任意节点的左节点小于该节点，右节点大于该节点，左右子树高度差不超过1。对于含有n个节点的AVL树和红黑树的查询、插入和删除的时间复杂度最坏都是log(n)。

为了防止二叉搜索树退化成链表，产生了平衡二叉树，AVL树是带有平衡条件的二叉查找树，一般是用平衡因子差值（左子树的高度减去右子树的高度）判断是否平衡并通过旋转（左旋、右旋）来实现平衡，左右子树高度差不超过1。和红黑树相比，AVL树是严格的平衡二叉树，平衡条件必须满足(所有结点的左右子树高度差不超过1)。不管我们是执行插入还是删除操作，只要不满足平衡条件，就要通过旋转来保存平衡，因为旋转非常耗时，由此可以知道AVL树适合用于插入与删除次数比较少，但查找多的情况。

**3. 红黑树**

(插入、删除、查找 O(logn))

红黑树是一种含有红黑结点并能自平衡的二叉搜索树。红黑树确保没有一条路径会比其他路径长出两倍，所以，红黑树是一种弱平衡二叉树。(由于弱平衡，在相同节点数情况下，AVL树的高度低于红黑树)，相对于要求严格的AVL树来说，它的旋转次数少，插入最多两次旋转，删除最多三次旋转，所以对于搜索，插入，删除操作较多的情况下，我们就用红黑树）。




红黑树也被称为特殊的平衡二叉搜索树，它不严格遵循平衡的条件，它遵循二叉搜索树的条件：

- 若左子树不空，则左子树上所有结点的值均小于或等于它的根结点的值。
- 若右子树不空，则右子树上所有结点的值均大于或等于它的根结点的值。
- 左、右子树也分别为二叉搜索树

  它还满足如下几点要求：
- 树中所有节点非红即黑。
- 根节点为黑节点，每个叶子节点为不存数据的黑色空节点(NIL节点)。
- 每个红节点的子节点必为黑（黑节点子节点可为黑，也就是说从根到每个叶子节点的所有路径上不能有两个连续的红色节点）。
- 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点

**4.红黑树和平衡二叉搜索树(AVL)区别?**

(为什么map底层要用红黑树实现而不用AVL实现)

主要从查找效率(平衡方面)，和插入以及删除节点方面考虑。

AVL树是严格平衡(左右树高差一)，而红黑树通过增加节点颜色从而实现**部分平衡**，插入节点两者都可以最多两次实现复衡，而删除节点，红黑树最多三次旋转就可实现复衡，旋转的量级是O(1)，而AVL树需要维护从被删除节点到根节点这几个节点的平衡，旋转的量级是O(logn)，所以对比删除节点操作的话，红黑树效率更高，但是因为红黑树是非严格平衡，它的高度最坏情况接近2log(n)，所以它的查找效率比AVL树低。(avl树高是logn，树越高查找效率越低)
(红黑树是功能、性能、空间开销的折中结果。)

总结：实际应用中，若搜索的次数远远大于插入和删除，那么选择AVL，如果搜索，插入删除次数几乎差不多，应该选择红黑树。

**5. 红黑树和哈希表的区别?**

(1)红黑树是有序的(map遍历时始终是按key的递增顺序)，哈希表是无序的
(2)红黑树占用的内存更小(仅需要为其存在的节点分配内存)，而哈希表需要预先分配足够的内存来存储包含映射关系的那个散列表，如果内存分配过大的话，可能会造成一些浪费。
(3)红黑树的查找、增删节点操作时间复杂度都是Olog(n)，哈希表的查找和增删节点操作在理论上时间复杂度都是O(1)，这个前提是不发生哈希冲突，在发生哈希冲突时，哈希表的插入和删除最坏能达到O(n)，(接下来可以说解决哈希冲突的办法)

**6. B+树与B树的区别?**

B+树非叶子节点不存数据只存索引，B树非叶子节点存储数据

B+树查询效率更高。B+树使用双向链表串连所有叶子节点，区间查询效率更高（因为 所有数据都在B+树的叶子节点，扫描数据库 只需扫一遍叶子结点就行了），但是B树 则需要通过中序遍历才能完成查询范围的查找。

B+树查询效率更稳定。B+树每次都必须查询到叶子节点才能找到数据，而B树查询的 数据可能不在叶子节点，也可能在，这样就会造成查询的效率的不稳定

B+树的磁盘读写代价更小。B+树的内部节点并没有指向关键字具体信息的指针，因此 其内部节点相对B树更小，通常B+树矮更胖，高度小查询产生的I/O更少。

# 1.vector

### 1.1 vector的底层实现

```
//是使用 3 个迭代器（可以理解成指针）来表示的
//_Alloc 表示内存分配器，此参数几乎不需要我们关心
template <class _Ty, class _Alloc = allocator<_Ty>>
class vector{
    ...
protected:
    pointer _Myfirst;
    pointer _Mylast;
    pointer _Myend;
};
```

其中，*Myfirst* 指向的是 *vector* 容器对象的起始字节位置；Mylast 指向当前最后一个元素的末尾字节；_myend 指向整个 vector 容器所占用内存空间的末尾字节。


vector底层是数组,它使用了三个迭代器，分别指向vector容器对象的起始字节位置，当前最后一个元素的末尾字节，整个vector容器所占内存空间的末尾字节，通过这三个迭代器vector可以实现首尾标识、容器大小、空容器判断等功能。

```
template <class _Ty, class _Alloc = allocator<_Ty>>
class vector{
public:
    iterator begin() {return _Myfirst;}
    iterator end() {return _Mylast;}
    size_type size() const {return size_type(end() -
begin());}
    size_type capacity() const {return
size_type(_Myend - begin());}
    bool empty() const {return begin() == end();}
    reference operator[] (size_type n) {return *
(begin() + n);}
    reference front() { return *begin();}
    reference back() {return *(end()-1);}
    ...
};
```

### 1.2 vector的扩容机制

(vector的大小是它实际所包含元素个数，vector的容量是指在不分配更多内存的情况下可保存的最多元素个数。)

当vector的大小和容量相等时(size==capacity)，也就是满载时，如果再向其中添加元素需要扩容，vector的扩容过程包括3步：

1.重新申请一块大小是当前vector容量两倍的空间(使用resize())。

2.将vector容器存储的所有元素，依照原来次序从旧存储空间复制到新的存储空间

3.析构掉(删除)旧存储空间中的所有元素，释放旧的存储空间。

1、弃用现有的内存空间，（使用resize()）重新申请更大的内存空间(1.5~2倍)；

2、将原来的数据，按原有顺序拷贝到新的内存空间中，如果是基本类型数据直接调用memcpy拷贝过去，复杂类型数据每个都调用拷贝构造函数进行初始化；

3、将旧空间释放

### 1.3 vector的clear和swap

vector的clear()操作只是清空vector的元素，而不会将内存释放掉,vector的swap操作会删除元素并且释放内存

### 1.3迭代器失效问题

#### 1.3.1 迭代器

STL中迭代器主要用来把容器和算法结合起 来，扮演容器与算法之间的胶合剂，它实际上是一个类模板，是对指针的封装，模拟了指针的一些功能，重载了指针的一些操作符，比如解引用*、指针指向操作符->、++、--等。

迭代器失效实际上就是指迭代器底层的指针所指空间被销毁了，如果继续使用失效的迭代器可能会造成程序崩溃。

#### 1.3.2 失效

迭代器失效主要包括两种类型，一是插入元素，二是删除元素：
a.插入元素，使容器中某些元素发生迁移，导致存放容器之前元素的空间失效，从而使指向原空间的迭代器失效。
b.删除元素，使容器中某些元素次序发生变化，原本指向某元素的迭代器不再指向它之前指向的元素。(这里的失效应该指迭代器和原元素的绑定关系发生了改变)

以vector为例：
**插入元素：**
1、尾后插入：size < capacity时，首迭代器不失效尾迭代失效（未重新分配空间），size == capacity时，所有迭代器均失效（需要重新分配空间）。
2、中间插入：中间插入：size < capacity时，首迭代器不失效但插

入元素之后所有迭代器失效，size == capacity时，所有迭代器均失效。

**删除元素：**
尾部删除：只有尾迭代失效。
中间删除：删除位置之后所有迭代失效。
deque 和 vector 的情况类似,
而list双向链表每一个节点内存不连续, 删除节点仅当前迭代器失效,erase返回下一个有效迭代器; map/set等关联容器底层是红黑树删除节点不会影响其他节点的迭代器, 使用递增方法获取下一个迭代器 mmp.erase(iter++);

·map，list，set容器中：删除，插入操作会导致指向该元素的迭代器失效，其他元素　迭代器不受影响。

### 1.4 vector添加元素(push_back()和emplace_back())

emplace_back() 和 push_back() 的区别，就在于底层实现的机制不同。push_back() 向容器尾部添加元素时，首先会创建这个元素，然后再将这个元素拷贝或者移动到容器中；而 emplace_back() 在实现时，则是直接在容器尾部创建这个元素，省去了拷贝或移动元素的过程（并且如果是右值的话empalce_back还会完美转发）。现在push_back()底层也是emplace_back()，所以使用push_back()已经习惯了。

### 1.5 vector插入元素（insert()和emplace())

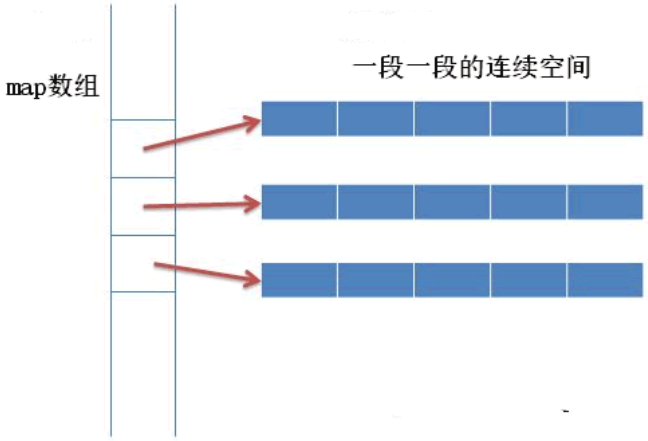
简单的理解，就是 emplace() 在插入元素时，是在容器的指定位置直接构造元素，而不是先单独生成，再将其复制（或移动）到容器中

## 2.deque

### 2.1 deque的底层实现

deque 容器(双端队列容器)存储数据的空间是由一段一段等长的连续空间构成，各段空间之间并不一定是连续的，可以位于内存的不同区域。

为了管理这些连续空间，deque 容器底层使用一个数组（数组名假设为 map）存储着各个连续空间的首地址。也就是这个数组中存储的都是指针，指向那些真正用来存储数据的各个连续空间。通过建立这个数组，deque 容器申请的这些分段的连续空间就能实现“整体连续”的效果。




换句话说，当 deque 容器需要在头部或尾部增加存储空间时，它会申请一段新的连续空间，同时在 map 数组的开头或结尾添加指向该空间的指针，由此该空间就串接到了 deque 容器的头部或尾部。(如果 map 数组满了怎么办？再申请一块更大的连续空间供 map 数组使用，将原有数据（很多指针）拷贝到新的 map 数组中，然后释放旧的空间。)

# 3.list

list容器的底层实现有的版本STL标准库是使用双向链表，内部通常包含两个指针，分别指向链表头部空白节点和尾部空白节点。而有的版本STL标准库使用的是双向循环链表，这样只需要一个指针就可以表示list容器的首尾元素。

# 4.map相关

## 4.1map实现原理

map内部实现了一个红黑树，存储的是键值对，红黑树有自动排序的功能，map内部所有元素都是有序的，红黑树的每一个节点都代表着map的一个元素。因此，对于map进行的查找、删除、添加等操作都相当于是对红黑树进行的操作。
map中的元素是按照二叉树搜索树存储的，特点就是左子树上所有节点的键值都小于根节点的键值，右子树所有节点的键值都大于根节点的键值。使用中序遍历可将键值按照从小到大遍历出来。

## 4.2unordered_map

unordered_map的底层实现是一个**哈希表**（也叫散列表），通过把关键码值映射到Hash表中一个位置来访问记录，查找时间复杂度可达O(1)，它在海量数据处理中有广泛应用。因此，元素的排列顺序是无序的。

## 4.3 map与unordered_map区别

map底层是红黑树实现，unordered_map底层是哈希表实现
map中元素是有序的(map内部默认按照key值递增排序)，unordered_map中元素是无序的

运行效率方面：unordered_map最高，而map效率较低但提供了稳定效率和有序的序列。

占用内存方面：map内存占用略低，unordered_map内存占用略高,而且是线性成比例的。

需要无序容器，**快速查找删除，不担心略高的内存**时用unordered_map；有序容器**稳定查找删除效率，内存很在意**时候用map。

### 4.4 map和set的区别

map和set都是关联式容器，底层实现都是红黑树，所以元素会自动排序，但不允许重复
map是以键值对的方式存储数据的，一个key对应一个value，set只存放键值，只有key 。set中的值不允许修改，map中的key不可以修改，但是key对应的value可以修改。

# 9.时间复杂度

### 9.1 STL常用容器时间复杂度

STL中常用的容器有vector、deque、list、map、set、multimap、multiset、unordered_map、unordered_set等。容器底层实现方式及时间复杂度分别如下：

1. vector

   采用一维数组实现，元素在内存连续存放，不同操作的时间复杂度为：

   插入: O(N)

   查看: O(1)

   删除: O(N)

2. deque

   采用双向队列实现，元素在内存连续存放，不同操作的时间复杂度为：

   插入: O(N)

   查看: O(1)

   删除: O(N)

3. list

   采用双向链表实现，元素存放在堆中，不同操作的时间复杂度为：

插入: O(1)

查看: O(N)

删除: O(1)

4. map、set、multimap、multiset

   上述四种容器采用红黑树实现，红黑树是平衡二叉树的一种。不同操作的时间复杂度近似为:

   插入: O(logN)

   查看: O(logN)

   删除: O(logN)

5. unordered_map、unordered_set、unordered_multimap、unordered_multiset

   上述四种容器采用哈希表实现，不同操作的时间复杂度为：
   插入: O(1)，最坏情况O(N)

   查看: O(1)，最坏情况O(N)

   删除: O(1)，最坏情况O(N)

# 1. map、set、multimap、multiset底层实现

这几种容器底层是用红黑树实现的，红黑树是一种平衡二叉搜索树，其左子树上的所有节点的值均小于根节点，右子树上所有节点的值均大于根节点的值，且左右子树的高度差不能大于1（这是二叉平衡搜索树的，红黑树是确保没有一条路径会比其他路径长出两倍）这样就保证了数存储的节点是有序的，查询效率及增删效率是OlogN

# 2.unordered_map、unordered_set、unordered_multimap、unordered_multiset底层实现

这几种容器的底层实现是哈希表。哈希表相关见0.6。查询及增删效率都是O1